Technical Report
CMU/SEI-91-TR-18
ESD-91-TR-18

Carnegie-Mellon University
Software Engineering Institute

Durra: A Task-Level Description Language Reference Manual (Version 3)

Mario R. Barbacci
Dennis L. Doubleday
Michael J. Gardner
Randall W. Lichota
Charles B. Weinstock

December 1991

The following statement of assurance is more than a statement required to comply with the federal law. This is a sincere statement by the University to assure that all people are included in the diversity which makes Carnegie Mellon an exciting place. Carnegie Mellon wishes to include people of [illegible] origin, sex, handicap, religion, creed, ancestry, belief, age, veteran status or sexual orientation.

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate [illegible] color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Education Amendments of 1972, Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders. In addition, Carnegie Mellon does not discriminate [illegible] the basis of religion, creed, ancestry, belief, age, veteran status or sexual orientation [illegible] ing application of this policy should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA [illegible] Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA [illegible]

Technical Report
CMU/SEI-91-TR-18
ESD-91-TR-18
December 1991

Durra: A Task-Level Description Language Reference Manual (Version 3)

Mario R. Barbacci
Dennis L. Doubleday
Michael J. Gardner
Randall W. Lichota
Charles B. Weinstock

Distributed Systems Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

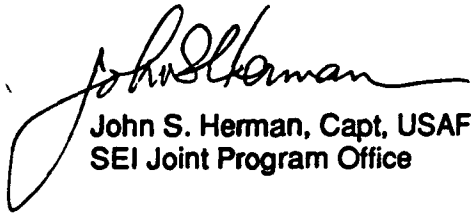
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



John S. Herman, Capt, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.
This report was funded by the U.S. Department of Defense.

Table of Contents

1 Introduction 1

2 A Development Scenario 3
2.1 Component Creation Activities 5
2.2 Application Creation Activities 5
2.3 Application Execution Activities 6

3 Notes on Syntax 7
3.1 Comments 7
3.2 Keywords and Predefined Identifiers 7

4 Names, Values, and Expressions 9
4.1 Local and Global Names 9
4.2 Literal and Non-Literal Values 10
4.3 Expressions 11

5 Compilation Units 13
5.1 Type Declarations 13
5.2 Task and Channel Descriptions 14

6 Task and Channel Selections 17

7 Interface Information 19

8 Behavioral Information 21

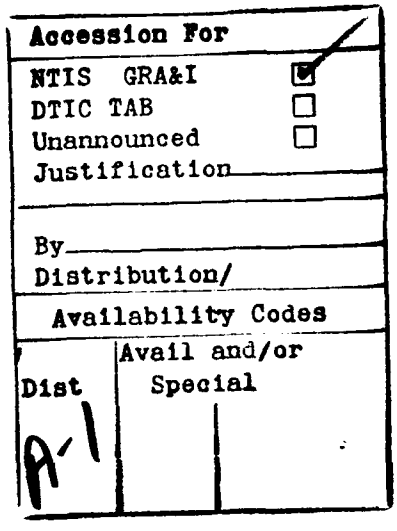
9 Attributes 23
9.1 Rules for Matching Selections with Descriptions 24
9.2 Soft- and Hard-matching between Selections and Descriptions 24
9.3 Global Attributes 25

10Structural Information 27
10.1 Component Declarations 27
10.2 Cluster Declarations 28
10.3 Structure Declarations 28
10.3.1 Baseline and Delta Components 29
10.3.2 Port Bindings 30
10.3.3 Component Connections 30
10.4 Reconfigurations 32

Appendix A Predefined Functions 35
A.1 current_atime, current_dtime, current_ptime 35
A.2 atime, dtime, ptime 35
A.3 Signal 36
A.4 Sizeof 36

Appendix B Predefined Attributes 37
B.1 Inheritance of Predefined Attributes 37
B.2 Processor Attribute 37
B.3 Package_name and Procedure_name Attributes 38
B.4 Process_Name Attribute 39

References 41



Durra: A Task-Level Description Language Reference Manual (Version 3)

Durra, also called "Indian millet" and "Guinea corn," is a type of grain sorghum with slender stalks, widely grown in warm dry regions. Durra sounds like "durable" which isn't a bad connotation. Carnegie Institute personnel provided a name denoting the very largest grain, but we respectfully declined their suggestion of "corn."

Abstract: Durra is a language designed to support the development of distributed programming applications consisting of concurrent, large-grained processes devoted to specific pieces of the application. During execution time the application processes run on possibly separate processors, and communicate with each other by sending messages of different types across communication links. The application developer is responsible for prescribing a way to manage all of these resources, called a task-level application description. It describes the processes to be executed, the assignments of processes to processors, and the communication channels required to transmit messages data between processes. Durra is a task-level description language, a notation in which to write these application descriptions.

This document is a revised version of the original reference manual.[1] It describes the syntax and semantics of the language and incorporates all the language changes introduced as a result of our experiences writing application descriptions in Durra.

A companion document, *Durra: A Task-Level Description Language User's Manual*, describes how to use the compiler and support tools.

1 Introduction

Many computation-intensive, real-time applications require efficient concurrent execution of multiple tasks as independent programs devoted to specific pieces of the application. Typical tasks include sensor data collection, obstacle recognition, and global path planning in robotics and vehicular control applications. Since the speed and throughput required of each task may vary, these applications can best exploit a computing environment consisting of multiple special and general purpose processors that are connected logically, though not necessarily physically. We call this environment a heterogeneous machine.

During execution time, application programs run on possibly separate processors and communicate by sending messages of different types. Since the patterns of communication can vary over time, and the speed of the individual processors can vary over a wide range, addi-

[1] M.R. Barbacci and J.M. Wing, *Durra: A Task-Level Description Language*. Technical report (CMU/SEI-86-TR-3, DTIC ADA178975), Software Engineering Institute, Carnegie Mellon University, December, 1986.

tional hardware resources in the form of communication networks, gateways, and data stores may be required in the heterogeneous machine.

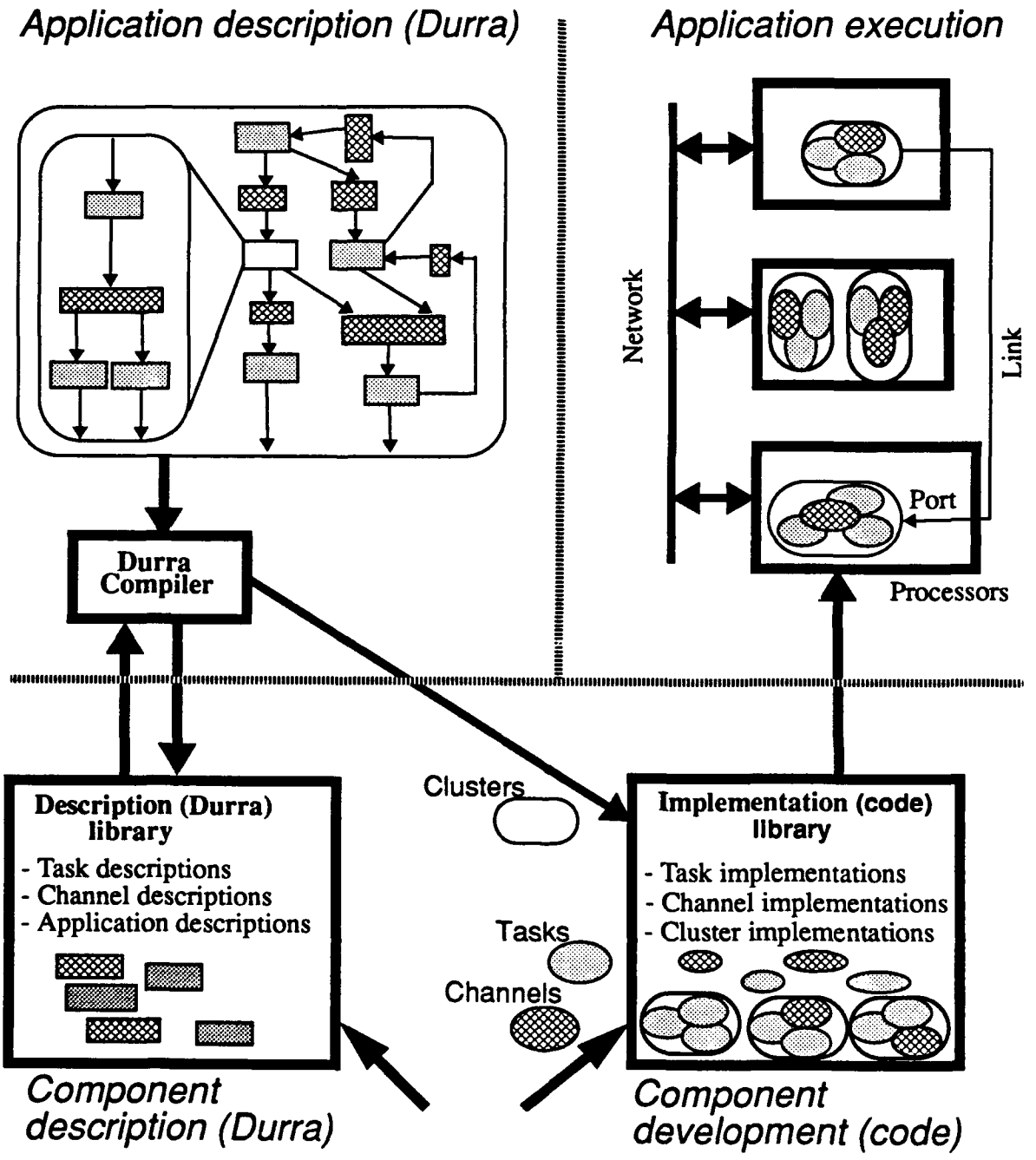
The application developer is responsible for prescribing a way to manage these resources. We call this prescription a task-level application description, which describes the component programs and the possible assignments of resources (processors, communication networks, et-cetera) to programs. A task-level description language is a notation in which to write these application descriptions. Durra is a task-level description language, a notation in which to write application descriptions.

Although Durra is hardware independent and can be used on a variety of processors and communication networks, we, for the purpose of explanation, will assume an abstract machine with multiple non-homogeneous processors, a local area network to allow direct communication between processors, and an operating system or runtime executive running on each processor and providing reliable communications facilities.

Actual experience with the implementation of the language and runtime environment and building demonstration applications has resulted in a number of changes to the language and these changes are reflected in the revised version of this language reference manual. The most notable changes are the elimination of the Durra runtime executive as a separate program in each of the network processors, and the replacement of FIFO queues by channels, a more general communication component. A channel provides a multiple-input, multiple-output communication link between application tasks, and can provide arbitrary routing, filtering, and storage disciplines for messages. In the new implementation the application developer can designate groups of processes (instances of tasks) and links (instances of channels) that are advantageous or necessary to execute as concurrent threads of a multi-threaded program. Message passing between components that are linked together in this fashion can be implemented more efficiently than with a general purpose network protocol such as TCP/IP. The gain in communication performance is obtained at the cost of sharing the processor and imposing coding conventions on the developers. We will refer to the multi-threaded program resulting from linking together several component as an application cluster, or cluster.

2 A Development Scenario

We see three distinct activities in the process of developing a distributed application using Durra: (a) the development of components (task/channel descriptions and implementations), (b) the development of an application description, and (c) the execution of the application.



Although we will describe the activities in this order, the actual development process might involve successive iterations over these steps. That is, an application is not necessarily developed bottom-up as the scenario might seem to imply. The developer could start with some gross level decomposition of the application into subsystems, and a prototype of the application might be developed with these high level components. Over time, these subsystems might be further decomposed into finer components and the steps repeated as appropriate. The scenario illustrated in the figure is just an illustration of the kinds of activities involved and is not a fixed prescription.

In describing these activities, we will use the following definitions:

channel description — a template written in Durra specifying the properties of a channel implementing a communication facility. Channel descriptions are compiled and stored in Durra libraries.

channel implementation — code written to implement a specific communication facility for application tasks. Channels are passive components that react to requests from tasks. Channel implementations are compiled and stored in object code libraries.

cluster — a group of tasks and channels linked together and executed as a multi-threaded program. The source code for the main program unit of a cluster is generated by the Durra compiler. Clusters are compiled and stored in object code libraries.

link — an instance of a channel providing communications between two or more processes.

port — a logical input or output device of a process or link. Input ports get messages from other components; output ports send messages to other components.

task description — a template written in Durra specifying the properties of a task implementing a piece of an application. Task descriptions are compiled and stored in Durra libraries.

task implementation — code written to implement a piece of an application. Tasks are active components that generate requests to the channels. Task implementations are compiled and stored in object code libraries

process — an instance of a task implementing part of an application. A process can execute as a single program (actually, a one-thread cluster) or as a thread within a multiple-thread cluster.

2.1 Component Creation Activities

In this phase, the developer defines the application components (tasks and channels) using domain specific knowledge. Some components might be domain specific such as sensor processing, map database management, route planning, etcetera. Other components might be of a more general nature, such as sorting, array operations, etcetera. An application component consists of a description and an implementation:

> A component implementation is an executable program. For a given task, there may be possibly many implementations, differing in programming language (e.g., C, Ada), processor type (e.g., Motorola 68020, DEC VAX), performance characteristics, or other attributes. The writing of a task implementation is more or less independent of Durra and involves the coding, debugging, and testing of programs on various machines. Available component implementations are stored in the appropriate object code libraries.

A component description is a template specifying properties of a component implementation, such as the types of data it produces or consumes, the ports it uses to communicate with other tasks, formal specifications of behavior, and other attributes of the implementation.

2.2 Application Creation Activities

In this phase, the developer writes an application description that specifies the desired components and their interconnection. Syntactically, an application description is similar to a task description and can be stored in the library as a new component task. This allows the developer to write hierarchical application descriptions. When the application description is compiled, the Durra compiler identifies library task and channel descriptions that meet selection criteria specified by the user and generates the cluster's main units.

As described earlier, component implementations are developed, tested, and stored in object code libraries independently of each other. To link groups of task and channel implementations into clusters, the Durra compiler generates the clusters' main units. In the current implementation, these main units are small Ada procedures that explicitly import the specific components of the cluster. Only these main units have to be compiled. The component implementations are retrieved from the object code library and linked with the appropriate main unit using a suitable Ada development environment. The main units constitute an additional thread of control within the cluster (i.e., they constitute an additional component process in the cluster, albeit not one specified by the application developer). These additional tasks will be referred to as the cluster managers. The application developer can specify how component tasks and channels are to be grouped into clusters. The extreme cases are: a) all components are linked together as one cluster, and b) each component is a separate cluster. The application developer can also specify what clusters run on each processor.

2.3 Application Execution Activities

To execute an application, the developer loads and starts the clusters in the appropriate processors. Each cluster's main unit contains information about the structure of the other clusters and the reconfigurations specified by the application developer. This information is used by the cluster manager to provide communication support for the local application processes.

Application processes communicate with each other through the same interface (i.e., procedure calls to the cluster manager) regardless of process location (i.e., processes within the same cluster, processes in clusters within the same processor, and processes in clusters in different processors communicate in the same fashion). The cluster manager implements the port operations (a subset of the interface calls) either as local procedure calls or remote procedure calls depending on the location of the communicating processes.

3 Notes on Syntax

To describe the syntax of Durra, we use standard Backus-Naur-Form (BNF), with the following conventions:

- Terminal symbols are enclosed in quotation marks ("and"), but the quotation marks do not belong to the terminal.
- No distinction is made between upper and lower case letters in terminals and non-terminals.
- A non-terminal of the form xyz_List$_{comma}$ stands for a list of one or more xyz's separated by commas
- Vertical bars ("|") separate alternative productions. Braces ("{"and "}") indicate optional components of a production.

3.1 Comments

Comments in a Durra description start with a double hyphen ("--"). Any text between the double hyphen and the end of the line is ignored.

3.2 Keywords and Predefined Identifiers

Keywords are highlighted in normal text by writing them in **bold face**. Predefined identifiers are highlighted in normal text by writing them inside quotation "marks".

The following words are keywords in the language: **and, attribute, attributes, baseline, begin, behavior, bind, channel, cluster, clusters, component, components, end, enter, exclude, exit, in, include, is, not, null, or, out, port, ports, previous, reconfiguration, reconfigurations, size, structure, structures, task, to, type, union, when.**

Several keywords exist in both singular and plural form and have the same meaning in either form: **attribute(s), channel(s), cluster(s), component(s), port(s), process(es), reconfiguration(s), and structure(s).**

The following words are predefined identifiers in the language: "atime", "current_atime", "current_dtime", "current_ptime", "dtime", "identifier", "integer", "null", "ptime", "package_name" "procedure_name", "process_name" "processor", "real", "signal", "sizeof", "string".

4 Names, Values, and Expressions

4.1 Local and Global Names

Syntax:

AttributeName ::= Identifier

ChannelName ::= Identifier

ClusterName ::= Identifier

LinkName ::= Identifier { "["IntegerRange "]" }

PortName ::= Identifier { "["IntegerRange "]" }

ProcessName ::= Identifier { "["IntegerRange "]" }

TypeName ::= Identifier

TaskName ::= Identifier

ComponentName ::= ProcessName |
LinkName

GlobalAttributeName ::=
{ ComponentName_List$_{period}$"." } AttributeName

GlobalLinkName ::= { ComponentName_List$_{period}$"." } LinkName

GlobalPortName ::= { ComponentName_List$_{period}$ "." } PortName

GlobalComponentName::=
ComponentName_List$_{period}$ "." } ProcessName

ParameterName ::= Identifier

SpecificationName := Identifier

StructureName ::= Identifier

Meaning:

Identifiers are sequences of letters, digits, and underscores ("_"). An identifier must begin with a letter and end with a letter or a digit. Consecutive underscores in the middle of an identifier are not allowed.

In a task or channel description the writer declares the names of ports, attributes, internal components, etcetera. These names are local to and unique within a task or channel description. Outside a description, these ports, attributes, and components are identified by a global name obtained by prefixing the name of a process or link (i.e., an instance of the task or channel) to the name of the port, attribute, or internal component. For example, `p1.out2` could be the name of port `out2` declared inside a task instantiated as process `p1`. If necessary, multiple process names as prefixes can be used. For example, `p1.p2.c` is the global name for some attribute `c` inside process `p2` which is inside process `p1`.

For convenience, the user can declare arrays of ports, links, and processes by specifying the name of the array and the range of indices for the array elements.

4.2 Literal and Non-Literal Values

Syntax:

IntegerLiteral ::= { "+" | "-" } sequence_of_digits

RealLiteral ::= { "+" | "-"} sequence_of_digits
{ "." {sequence_of_digits} }

StringLiteral ::= "" {sequence_of_characters} """"{"&" StringLiteral}

IntegerValue ::= IntegerLiteral |
ParameterName |
GlobalAttrValue |
FunctionCall

RealValue ::= RealLiteral |
ParameterName |
GlobalAttrValue |
FunctionCall

StringValue ::= StringLiteral |
ParameterName |
GlobalAttrValue |
FunctionCall

FunctionCall ::= FunctionName { ""Expression_List$_{comma}$ ")" }

Value ::= IntegerValue |
RealValue |
StringValue |
"(" ArithmeticExpression ")"

Meaning:

Integer and real numbers are always decimal (i.e., base 10). A real number can terminate with a period (".") without a fractional part.

Strings are sequences of ASCII printable characters, enclosed in double quotation marks ("). A double quotation mark inside a string must be written as two consecutive double quotation marks:

" A string with a double quotation mark, "", inside "

Literals denote constants of the appropriate type. An IntegerLiteral is a non-empty sequence of digits. A RealLiteral is also a non-empty sequence of digits with an embedded decimal point. A StringLiteral is a (potentially empty) sequence of characters within quotation marks.

Each of the non-terminals IntegerValue, RealValue, and StringValue stands for (a) literals (constants) of the appropriate kind, or (b) names of parameters (Chapter 5.2) or attributes (Chapter 9) whose values are of the appropriate kind, or (c) calls to predefined functions (Appendix A) returning values of the appropriate kind.

4.3 Expressions

Syntax:

Expression	::= { **not** } Term { **or** Expression }
Term	::= Relation { **and** Term }
Relation	::= ArithmeticExpression { RelOp ArithmeticExpression }
RelOp	::= "=" | "/=" | ">" | ">=" | "<" | "<="
ArithmeticExpression	::= { "+" | "-" } Factor { "+" | "-" Factor }
Factor	::= Value { "*" | "/" Value }
IntegerExpression	::= ArithmeticExpression
IntegerRange	::= IntegerExpression { ".." IntegerExpression }

Meaning:

Expressions can be used to specify actual parameters and attribute values in a component selection (Chapter 6). These expressions are evaluated at compile time and their values are used by the Durra compiler to identify component descriptions in the library.

Expressions can be used to specify reconfiguration conditions in a task description (Section 10.4). These expressions are evaluated at execution time. These values are used by the cluster managers to trigger dynamic reconfigurations of the application.

The production IntegerExpression is an arithmetic expression that permits only integer values in some contexts. For example, the bounds of an array of ports must be an integer value.

5 Compilation Units

There are three kinds of compilation units (i.e., separately compilable units): type declarations, task descriptions, and channel descriptions.

Syntax:

CompilationUnit ::= TypeDeclaration |
TaskDescription |
ChannelDescription

Meaning:

Each compilation unit must be submitted to the Durra compiler as a single text file. If no errors are detected, the unit is entered into the library.

Type declarations are treated like the other compilation units for the purposes of the development scenario (Chapter 2). That is, a type declaration is treated as an application "component" that must be described (i.e., declared), compiled, and entered into the library before it can be used in larger components or applications.

5.1 Type Declarations

Syntax:

TypeDeclaration ::= **TYPE** TypeName **IS** ScalarStructure ";" |
TYPE TypeName **IS** UnionStructure ";"

ScalarStructure ::= **SIZE** IntegerRange |
SIZE "*"

UnionStructure ::= **UNION** "("TypeName_List$_{comma}$ ")"

Examples:

```
type packet is size 128.. 1024;
        -- Packets are of variable length
type four_packets is size 4 * sizeof(packet);
        -- Type consisting of four packets
type mix is union (heads, tails);
        -- Mix data could be heads or tails
```

Meaning:

Type declarations are compilation units that define the structure of the data produced or consumed by the application tasks. A type declaration introduces a global name for a data type, or a set of previously declared types, which can then be used in port declarations.

There are two kinds of type declarations. The simpler data type (**SIZE**) is a sequence of bits of fixed or variable length. The length is specified by an integer value (the length is fixed), an integer range (the length varies between two boundaries), or "*" (the length is unknown).

More complex types are declared as a union (**UNION**) of a number of previously declared types where data items moving through a process or link port could be one of any of the member types.

Durra type declarations provide hints about the size of the messages transmitted. The channel implementations can use this information to allocate storage for messages. No semantic information is derived from the type declaration and the cluster manager will not attempt to transform the data to hide data layout differences between different languages or processor architectures.

5.2 Task and Channel Descriptions

Syntax:

```
TaskDescription        ::=SimpleTaskDescription |
                          CompoundTaskDescription

SimpleTaskDescription::=
                          TASK TaskName
                            {FormalParameterPart}
                              {InterfacePart}
                                {BehaviorPart}
                                  {AttributeDescPart}
                                        END TaskName ";"

CompoundTaskDescription::=
                        TASK TaskName
                          {InterfacePart}
                            {BehaviorPart}
                              {AttributeDescPart}
                                {StructurePart}
                                  END TaskName ";"
ChannelDescription   ::= CHANNEL ChannelName
                          {FormalParameterPart}
                            {InterfacePart}
                              {BehaviorPart}
                                {AttributeDescPart}
                                  END ChannelName ";"
FormalParameterPart ::=
                        '("FormalParameter_List comma ")"
FormalParameter      ::= Identifier ":" ParameterType
ParameterType        ::= "INTEGER" |
                         "REAL" |
                         "STRING" |
                         "IDENTIFIER"
```

Meaning:

Task descriptions are compilation units that define the properties of task implementations (i.e., user programs). Task descriptions are used as building blocks for application and compound-task descriptions.

Channel descriptions are compilation units that define the properties of channel implementations. Channel descriptions are used as building blocks for application-descriptions and compound task-descriptions.

A component description has a number of sections, most of which are optional: formal parameters, interface information, behavioral information, attributes, and structural information.

The parameter part consists of a list of typed parameter names used to customize a component description. Formal parameters are not allowed in a compound task description (i.e., only simple tasks and channels can be parameterized). Formal parameter names can be used in the component description anywhere a value of the appropriate type is allowed. The actual parameter value is specified during the task or channel instantiation. The parameter types (`INTEGER`, `REAL`, `STRING`, and `IDENTIFIER`) are built into the language and can not be specified by the user.

The interface part consists of a number of unidirectional, typed port declarations. The port types are not built into the language, and they must be declared by the user.

The behavior part consists a list of name/value pairs denoting formal properties of the component.

The attribute part consists of a list of name/value pairs denoting miscellaneous properties of the component. Attributes play a central role in identification and selection of components.

The structural information describes the internal components of a compound task. This is the only distinction between a simple and a compound task. Structural information is not allowed in a channel description (i.e., there are no compound channels). The structural information is divided into three sections. The component section enumerates a list of task and channel instantiations used as internal components. The structure section describes how these internal components are connected to each other to form a configuration. There can be one or more of these configurations in a structure section. The reconfiguration section describes conditional transitions between alternative configurations.

6 Task and Channel Selections

Syntax:

TaskSelection ::=
TASK *TaskName* {ActualParameterPart} ";" |
TASK TaskName {ActualParameterPart}
AttributeSelPart
END *TaskName* ";"

ChannelSelection ::=
CHANNEL *ChannelName* {ActualParameterPart} ";" |
CHANNEL *ChannelName* {ActualParameterPart}
AttributeSelPart
END *ChannelName* ";"

ActualParameterPart ::=
"("Expression_Listcomma ")"

Meaning:

Task and channel selections are templates used to identify and retrieve task and channel descriptions from the library. A given task or channel might have a number of different implementations that differ along dimensions such as algorithm used, code version, performance, or resources required. In order to select among a number of alternative implementations, the user provides a task or channel selection as part of a component declaration, as described in Section 10.1. This selection lists the desirable features of a suitable implementation.

The actual parameter values (expressions) substitute the corresponding formal parameters in the matching task or channel description anywhere the formal parameter identifiers occur in the description. These substitutions take place after the description and the selection are matched. Therefore, actual parameters cannot be used to identify a matching description in the library.

The attribute selection part is an expression of attribute values evaluated in the context of the attribute values provided in the task or channel description.

For brevity, if the attribute selection part is not provided, the closing "end *name*" is not required.

7 Interface Information

Syntax:

InterfacePart ::= **PORT** PortDeclaration_List$_{semicolon}$ ";"

PortDeclaration ::= PortName_List$_{comma}$ ":" **IN** TypeName |
PortName_List$_{comma}$ ":" **OUT** TypeName

Examples:

```
ports
   in1: in heads;
   out1, out2: out tails;
```

Meaning:

The interface information describes the ports of the component tasks or channels. A port declaration specifies the direction of the data movement and the type of data moving through the port.

8 Behavioral Information

Syntax:

BehaviorPart	::=	**BEHAVIOR** Specification_List $_{\text{Semicolon}}$ ";"
Specification	::=	SpecificationName "=" SpecificationValue
SpecificationValue	::=	identifier | value

Meaning:

The behavior part of a component description consists of one or more formal or informal behavioral specifications of the component. Each specification consists of a name and a value. There may be as many specifications as desired. The syntax and semantics of specification values are specification dependent. During compilation, these values are treated as uninterpreted identifiers, numbers, or strings, as the case may be. The Durra language simply provides a placeholder to capture behavioral specifications. It is up to the users to invoke appropriate tools to interpret the specification values. For example, behavioral specifications could be used to drive emulators, code generators, theorem provers, or other design tools. Consult the appropriate documentation for details about supported specifications.

9 Attributes

Syntax:

AttributeDescPart ::= **ATTRIBUTES** Attribute_List$_{semicolon}$ ";"

Attribute ::= AttributeName "=" Value |
AttributeName "=" ""Value_List$_{comma}$ ")"

AttributeSelPart ::= **ATTRIBUTES** Expression ";"

Examples:

```
        -- Attributes in a task declaration
attributes
   author = "jmw";
   color = ("red", "white", "blue");
   implementation = "cowcatcher";
   queue_size = 25;

        -- Attributes in task selections
attributes
   processor = Sun;
attributes
   author = "jmw" or author = "mrb";
attributes
   (color="red" or color="blue") and (author= Mr_Ed);
```

Meaning:

Attributes specify miscellaneous properties of a component. They are a means of indicating pragmas or hints to the Durra compiler and the cluster managers. In a task or channel description, the developer of a component lists the component properties and their possible values; in a task or channel selection, the user of a component writes an expression listing the desired name and values of the component's properties. Attribute values used in matching selections with descriptions must be constants, computable before execution time, i.e., components and their implementations are static properties of an application.

Example attributes include: author, version number, task implementation, and processor type. There may be as many attributes as desired. The syntax and semantics of the attribute values are attribute dependent. If the attribute is not predefined in the language, the values are treated as uninterpreted identifiers, numbers, or strings, as the case may be, and compatibility is based on value equality. If the attribute is predefined in the language, compatibility is attribute dependent. See Appendix Appendix B for details about the predefined attributes.

Component selection attributes are "merged" with component description attributes to define the set of attributes visible to the implementations and the cluster managers at runtime. See Appendix B for special inheritance rules in the case of predefined attributes (inheritance of user-defined attributes occurs only between a task selection and its task description.)

9.1 Rules for Matching Selections with Descriptions

If a selection includes an attribute expression (a predicate), a matching description must provide values that satisfy the predicate, i.e., the expression yields true when evaluated in the context of the values declared for the attribute.

If a description provides a list of values for an attribute, a matching selection is satisfied if any of these values satisfies the expression. For example, if a description includes an attribute `size` with values (1, 3, 5) then a selection with an attribute expression of the form `size > 4` is satisfied by the description because the value 5 makes the expression true.

If a selection provides a list of values, (e.g., `color /= (red, green)`) then all of the values in the selection must be satisfied (i.e., the color attribute in the description must not include `red` nor `green`).

If a selection includes an attribute expression that uses attributes not present in a description or if a description provides attributes not used in the attribute expression of a selection, these attributes are ignored for the purposes of matching a library component.

9.2 Soft- and Hard-matching between Selections and Descriptions

A selection can match multiple descriptions depending on what selection attributes can be ignored because they do not appear in a given description. The following example illustrates the situation.

Assume that the Durra library contains two tasks that share the same name (`t`) but differ in the names or values of various attributes:

```
task t ...attributes color = "red"; version = 1;...end t;

task t
...
   attributes color= "green"; version= 1; author= "mrb";
...
end t;
```

The following task selections fail to yield unambiguous matches:

```
p1: task t attributes version= 1; end t;
p2: task t attributes version= 1 and author="mrb"; end t;
```

Process `p1` fails because both task descriptions in the library match the information provided (task name `t` and attribute `version = 1`) and there is not reason to prefer one over the other. Process `p2` fails because both task descriptions match the selection. The second declaration of task `t` matches the selection because it has the same attribute names and values as those of the task selection; the first declaration of task `t` matches because it does not have an `author` attribute and under the rules of the language, the selection `author` is ignored.

Clearly there is a difference between these two situations. In the case of process `p1` all the information available in the selection was used but was not enough to distinguish between the available task descriptions. In the case of process `p2` one of the matches ignored part of the

information available in the selection while the other match used all of it. The Durra compiler handles these two situations differently.

When a match uses all the attributes available in a selection we refer to it as a "hard attribute match". When a match ignores some of the attributes available in a selection we refer to it as a "soft attribute match". Multiple hard matches are reported as serious errors (the compilation fails). One hard match and one or more soft matches are reported as minor errors: the compiler selects the hard matching task description. Multiple soft matches are reported as minor errors: the compiler selects one of the matching descriptions at random. In the absence of serious errors, a compilation does not fail solely because of minor errors. Nevertheless, the users should correct the selection templates to remove the ambiguities.

9.3 Global Attributes

The name of an attribute can appear in any context in which its value can appear. For instance, if the user defines an attribute `queue_size` with an integer value, as in the examples above, then the identifier `queue_size` can appear anywhere an integer value is expected. This permits the user to define families of components, i.e., components that share the same value for some attribute, as shown in the following example:

```
task Bedroom
   attributes shape = "hexagon"; size = 25;
end Bedroom;
task Bedroom
   attributes shape = "square"; size = 30;
end Bedroom;
task Kitchen attributes size = 25; end Kitchen
task Kitchen attributes size = 30; end Kitchen;
```

Assuming that the library contains the task descriptions shown above, the following task selections would select the version of task `Bedroom` with `shape ="hexagon"` (and `size = 25`) and the version of task `Kitchen` with `size=25` because this is the size of the selected bedroom:

```
B: task Bedroom attributes shape= "hexagon"; end Bedroom;
K: task Kitchen attributes size = B.size; end Kitchen;
```

10 Structural Information

Syntax:

StructurePart ::= ComponentClause
{ClusterClause}
StructureClause
{ReconfigurationClause}

ComponentClause ::= **COMPONENTS**
ComponentDeclaration_List$_{semicolon}$ ";"

ClusterClause ::= **CLUSTERS** ClusterDeclaration_List$_{semicolon}$ ";"

StructureClause ::= **STRUCTURES**
StructureDeclaration_List$_{semicolon}$ ";"

ReconfigurationClause::=
RECONFIGURATIONS
Reconfiguration_List$_{semicolon}$ ";"

Meaning:

The structural information of a compound task consists of four parts: a declaration of the components (i.e., instances of internal tasks and channels), the various ways in which these components can be connected to form alternative structures, the conditions under which changes in structure take place, and finally, the ways these components should be grouped together in clusters.

Channels and tasks are similar in many respects. Channel and task implementations are stored in code libraries; channel and task descriptions are stored in Durra libraries; channels and tasks are instantiated as components and connected in arbitrary structures. However, channels and tasks have fundamentally different behaviors. Tasks are active components (i.e., they initiate requests for sending or receiving messages.) Channels are passive components (i.e., they react to requests for sending or receiving messages.) This difference imposes restriction on the allowed connections between link and process ports to avoid deadlocks. If two channel ports are connected, neither would initiate a message passing operation; if two task ports are connected, neither would react to a request for a message passing operation initiated by the other task.

10.1 Component Declarations

Syntax:

ComponentDeclaration ::=
ProcessName_List$_{comma}$ ":" TaskSelection |
LinkName_List$_{comma}$ ":" ChannelSelection

Examples:

```
p[1..3], px: finder;
l: merge (nports, msg_type);
```

Meaning:

Component declarations specify the building blocks (internal links and processes) of a compound task. An instance of a task or a channel is bound to each component name in the list.

10.2 Cluster Declarations

Syntax:

ClusterDeclaration ::= ClusterName ":"
ClusterComponentName_List$_{comma}$

ClusterComponentName::=
ComponentName |
ClusterName

Examples:

```
clusters
   c1: p1, l1;
   c2: p2;
   -- Group components p1 and l1 as cluster c1.
   -- Cluster c2 consists of a single component, p2.
```

Meaning:

A cluster declaration is a directive to the Durra compiler. It is used to specify groups of component processes and links that must be linked together as a single program.

Component (processes and links) that are assigned to different clusters of a compound task can not be merged into one cluster in a larger task that includes instances of the compound task.

If a component process is an instance of a compound task and the corresponding task declaration specified cluster declarations, the names of the component clusters must be used instead of the name of the process (this follows from the restriction in the previous paragraph). Clusters can "grow" by naming them as components of a larger cluster (i.e., a cluster of an enclosing task) provided sibling clusters are never merged.

Cluster declarations specify a physical binding of components. They do not specify or suggest any connectivity between the components of a cluster. The logical binding of components is specified through structure declarations (Section 10.3).

10.3 Structure Declarations

Syntax:

StructureDeclaration ::=
StructureName ":"
BEGIN
{BaselineComponents}

{ExternalPortBindings}
{ComponentConnection_List$_{semicolon}$ ";"}
{NestedStructureDeclaration_List$_{semicolon}$ ";"}
END StructureName

NestedStructureDeclaration::=
StructureName ":"
BEGIN
{DeltaIncludeComponents}
{DeltaExcludeComponents}
{ExternalPortBindings}
{ComponentConnection_List$_{semicolon}$ ";"}
{NestedStructureDeclaration_List$_{semicolon}$ ";"}
END StructureName

Meaning:

A structure declaration specifies the (subset of) internal components to be used in a particular configuration of a compound task and how these components are connected to each other. To avoid repeating entire structure declarations when alternative configurations differ only in a few details, structure declarations can be nested and only the changes to the set of components currently in use need to be specified in the inner structure. Each structure declaration has a name. These names must be unique throughout the entire task description (i.e., nesting of structure declarations is not a name scoping mechanism).

Structure declarations specify a logical binding of components. They do not specify or suggest the grouping of components in clusters. The physical binding of components is specified through cluster declarations (Section 10.2).

10.3.1 Baseline and Delta Components

Syntax:

BaselineComponents ::=
BASELINE ComponentName_List$_{comma}$ ";"

DeltaExcludeComponents::=
EXCLUDE ComponentName_List$_{comma}$ ";"

DeltaIncludeComponents::=
INCLUDE ComponentName_List$_{comma}$ ";"

Meaning:

The list of components used in a top level structure declaration constitutes a baseline configuration. These components are listed following the keyword **BASELINE.**

For convenience, if all the components are considered to be part of the configuration then the baseline list can be omitted.

A list of components listed in a nested structure declaration constitutes a modification to the enclosing baseline configuration. A modification could be a list of components to be removed

from (**EXCLUDE**), or added to (**INCLUDE**) the enclosing baseline. The result is the baseline configuration for any inner structure declarations.

The ability to define structures that differ only in minor details, through modifications to an enclosing baseline structure, is provided as a convenience to the user. All structures could be specified as "top level" i.e., non-nested structures, using the baseline construct to list all the component without regard for what components any other structure might use. Nesting of structures is just a way to simplify writing lists of components. There is no other meaning attached to the nesting.

10.3.2 Port Bindings

Syntax:

ExternalPortBindings ::= **BIND** PortBind_List$_{comma}$ ";"

PortBind ::= PortName "=" PortName

Meaning:

When a task description specifies internal processes, the structure declarations must specify which of the internal process ports implements a port of the outer task. This is specified via a **BIND** declaration.

The left hand side of a port bind declaration is the name of an outer task port. The right hand side of a port bind declaration is the name of an inner process port. The bindings between internal and external ports are not permanent and can be replaced in alternative or nested structures.

Notice that port bindings are restricted to tasks and instances of tasks (i.e., processes) only. A port of a compound task cannot be implemented by a port of an internal channel.

10.3.3 Component Connections

Syntax:

ComponentConnection::=
LinkName ":"
SourcePortName_List$_{comma}$
">>"
DestinationPortName_List$_{comma}$

SourcePortName ::= GlobalPortName | **NULL**

DestinationPortName ::= GlobalPortName | **NULL**

Examples:

```
l1: p1.output > > p2.input;
   -- Link l1 is used to connect an output port of process
   -- p1 to an input port of process p2.
   -- l1 must be an instance of a channel and p1, p2
```

```
   -- must be instances of tasks

13: p1.output > > NULL;
   -- a link used to connect an output port of process p1
   -- to a NULL port.
```

Meaning:

A component connection specifies a link, a set of source ports, and a list of destination ports. The link transmits data from the source ports to the destination ports. The connection remains in effect while the application retains its structure. However, the same components could be connected in a different way in an alternative structure definition.

The source ports, listed to the left of the ">>", must be compatible with the link input ports. That is, source ports must be the output ports of some components and their types must be compatible with the types of the link input ports. Similarly, the destination ports, listed to the right of the ">>", must be compatible with the link output ports. That is, they must be declared as the input ports of some components and their types must be compatible with the types of the link output ports.

In principle, the source and destination ports named in a component connection should always be process ports, never link ports. However, it is often convenient to describe application structures in which links feed messages to other links. Strict adherence to the rules would require instantiating auxiliary, intermediate processes as the active components between two link ports. These processes would have exactly one input port and one output port and their behavior would be to loop receiving messages from the input port and transmitting them through the output port.

The Durra compiler simplifies the writing of an application description by providing implicit declarations of these auxiliary processes. These are instances of a "generic" task, parameterized with the appropriate message type. These processes, however, are not visible to the application developer and can not be named in baselines, cluster declarations, etc. Thus, it is legal to include link ports in the source and destination lists of a component connection statement, provided of course that the port directions and types are otherwise compatible.

The matching of source and destination ports with the link ports depends on the order in which the link ports are declared. The first source port must be type-compatible with the first link input port, the second source port must be type-compatible with the second link input port, and so on. All the link ports must be accounted for in this fashion.

The predefined name **NULL** can be used as a source or destination port for unused link ports. This feature should be used with care because an application task could be inadvertently blocked if it tries to receive data from a **NULL** port or send data to a **NULL** port. There is no process or link at the other end.

Nonunion types are compatible if they have the same name. A union source type requires a union link type and, in addition, the source type must be a subset of the corresponding link type. A union destination type must be a superset of the corresponding link output type. A nonunion source type must be equal to or, a subset of the corresponding link type. A nonunion destination type requires a nonunion link type and both types must have the same name.

Intuitively, these rules simply state that a message arriving on a source port can be received by the corresponding link input port, and that a message leaving a link output port can be received by the corresponding destination port.

10.4 Reconfigurations

Syntax:

```
Reconfiguration   ::= FromStructure "=>" ToStructure
                                  WHEN Expression |
                      ENTER "=>" StructureName {WHEN Expression}
FromStructure     ::= StructureName |
                      "*"
ToStructure       ::= StructureName |
                      EXIT |
                      PREVIOUS |
                      ENTER
```

Meaning:

A reconfiguration statement is a directive to the cluster managers. It is used to specify changes in the current structure of the application and the conditions under which these changes take effect. The reconfiguration condition is a Boolean expression involving signals from application components and information available to the cluster managers at run time.

If there is only one structure defined and there is no reconfiguration condition to delay the start of the application, the reconfiguration statement can be omitted.

The reconfiguration conditions are tested by the cluster managers concurrently. If more than one of these conditions becomes true at the same time, the cluster managers select one of the reconfiguration statements. This choice of alternative structure is non-deterministic.

The predefined structure name **ENTER** represents the empty, componentless structure before the start of the application. There can be any number of reconfiguration statements specifying **ENTER** as the previous structure. The conditional expression in a reconfiguration statement specifying the initial application structure is optional. This is the only case in which the conditional expression can be omitted.

If a reconfiguration condition is specified for the initial application structure, the start of the application is delayed until the condition is met. If there are several of these reconfiguration statements and more than one of the reconfiguration conditions becomes true at the same time, the cluster managers select one of the reconfiguration statements. This choice of initial structure is non-deterministic.

The predefined structure name **EXIT** represents the empty structure at the end of the application. There can be any number of reconfiguration statements specifying **EXIT** as the next structure.

The predefined structure name **PREVIOUS** represents the structure before the application adopted the current structure. There can be any number of reconfiguration statements

specifying **PREVIOUS** as the next structure. It is a convenient way to specify the return to a previous configuration after a temporary structure change.

If the current structure is the initial configuration, a return to the previous configuration is a return to the empty, componentless structure denoted by the predefined structure name **ENTER**.

The symbol " * " represents the set of all structure names and is a shorthand for specifying a transition from any structure to the same next structure. There can be any number of reconfiguration statements specifying " * " as the previous structure.

Appendix A Predefined Functions

Syntax:

```
FunctionName    ::= current_dtime |
                    current_atime |
                    current_ptime "("GlobalComponentName ")" |
                    atime "("ArithmeticExpression ")" |
                    dtime "("ArithmeticExpression ")" |
                    ptime "("GlobalComponentName ","
                                        ArithmeticExpression ")" |
                    signal "("GlobalComponentName ","
                                        ArithmeticExpression "')" |
                    sizeof "("TypeName ")"
```

Meaning:

The predefined functions are used to build expressions involving time values and type sizes (`sizeof`). Calls to the predefined functions can appear anywhere a value of the same kind as the return value can appear.

The values of time returned by the various functions described in this section are generated by a "global clock" implemented by a suitable synchronization protocol. Which protocol is used is an implementation detail that will vary from system to system. Application developers should consult the appropriate implementation notes for additional details.

Example:

```
current_ptime(p1)
   -- The amount of time component p1 has executed.
type msg is size 4 * sizeof(block);
   -- The size of a new type (msg) is a function of the
   -- size of a previously declared type (block).
```

A.1 current_atime, current_dtime, current_ptime

The function `current_atime` returns the number of seconds (a real value) from the start of the application.The function call `current_dtime` returns the number of seconds from the start of the current day (returns a value between 0.0 and 86,400.0). The function `current_ptime`(*component_name*) returns the elapsed time in seconds since the start of a given component (process or link) in the current configuration.

A.2 atime, dtime, ptime

The function atime(*duration*), dtime(*duration*), and ptime(*component_name*, *duration*) return the value or point in time *duration* seconds after the start of the application, the start of the current day, and the start of a component in the current configuration, respectively. The parameter to dtime can exceed 86,400.0 when computing future values of time.

A.3 Signal

The function `signal(`*component_name, integer_value*) returns true if the last signal raised by a component had a given value and returns false if no signals have been raised so far by the component or if the last signal raised had a different value.

Signal values are integer numbers that have no predefined meaning in the language. It is up to the application and component developers to adopt the appropriate interpretation and convention for signal values. See the *Durra User's Manual* for details about signal raising and other means of communication between the application components (processes and links) and the cluster managers.

A.4 Sizeof

The function call `sizeof(`*type_name*) is evaluated at compile time and returns the size, in bits, of the specified Durra type name in IntegerRange format:

```
type Byte is size 8;
type Stream is size *;
type My_Record is size 20.. 30;
type Four_Bytes is size 4 * sizeof(Byte);
type Byte_String is union (Byte, Stream);

sizeof(Byte) => 8
sizeof(Stream) => *
sizeof(My_Record) => 20.. 30;
sizeof(Four_Bytes) => 32;
sizeof(Byte_Stream) => *
```

Appendix B Predefined Attributes

The following attributes are predefined in the language: "package_name", "procedure_name", "process_name", and "processor".

B.1 Inheritance of Predefined Attributes

As described in Section 9, the component selection attributes are merged with the component description attributes to define the set of attributes visible to the component implementation at runtime. In addition to the merging of attributes between component selections and descriptions, predefined attributes are also inherited through multiple levels of component declarations, as illustrated in the following example:

```
-- Primitive components
task t11
   attributes processor = "y"; cluster = "c1";
end t11;

task t12 attributes cluster = "c2"; end t12;

-- Intermediate component
task t components p11: task t11; p12: task t12; end t;

-- Top level description
task example
   components
      p: task t attributes processor = "x"; end t;
end example;
```

Process *P* in the top level description is declared as an instance of task *T*, and the task selection specifies a value for the processor attribute ("x"). This value of the attribute is used as a default processor attribute in all processes and links declared inside *T* and these components in turn pass the attribute down to their internal processes and links.

Inheritance chains terminate when the inner component has an explicit definition of the attribute. In this example, process `p.p11`, an internal components of task `t` will have processor attribute "y" while process `p.p12`, the other component of task `t` will inherit processor attribute "x".

B.2 Processor Attribute

Syntax

ProcessorAttribute ::= "PROCESSOR" "=" StringValue

Examples:

```
Processor = "sun";
```

```
Processor = "sun1";
```

Meaning

The value of the processor attribute is the logical name of a processor or processor class on which a component implementation can execute

The value of the processor attribute can vary in specificity by using a processor class name or an individual processor name. For example, SUN could mean any SUN processor; SUN1 could mean a specific SUN processor. If the user specifies the name of a class of processors as the value of the processor attribute, any member of the class can be used to execute the component.

Since cluster declarations (Section 10.2) determine how components are grouped together into a single program, the processor attribute of the components, if specified, must be identical. The Durra compiler will issue an error message if a cluster would contain components meant to execute of different processors.

B.3 Package_name and Procedure_name Attributes

Syntax:

Package_NameAttribute::=
"PACKAGE_NAME" "=" StringValue

Procedure_NameAtribute::=
"PROCEDURE_NAME" "=" StringValue

Examples:

```
package_name = "broadcast";
```

Meaning:

The value of the package_name attribute is the name of the Ada package that implements a channel. The value of the procedure_name attribute is the name of the Ada procedure that implements a task. In both cases this is the Ada unit name, not the file name. These two attributes are mutually incompatible. At most one of them must be provided for a component:

1. Channels are always implemented as Ada packages and a link (instance of a channel) must specify a package_name attribute either in the channel selection or the channel description.
2. Simple tasks are always instantiated as Ada procedures and a process instantiated from a simple task must specify a procedure_name attribute (either in the channel selection or the channel description).
3. Compound tasks do not have implementations and a process instantiated from a compound task can not have either attribute.

Although the location and naming conventions of the Ada libraries may vary with the users environment, only the names of the units implementing the tasks and channels are needed for the Durra compiler to generate the cluster's main unit.

The steps necessary to compile and link the various units and clusters are Ada implementation specific and outside the scope of this manual. See the *Durra User's Manual* for additional details.

B.4 Process_Name Attribute

Syntax:

Process_NameAttribute::=
"PROCESS_NAME" "=" StringValue

Examples:

```
process_name = "lan[3].simulator[4].pdu_generator";
```

Meaning:

The value of the process_name attribute is an arbitrary string that can be used by a task implementation to identify itself. For example, it could be used to differentiate terminal output printed by concurrent tasks.

This attribute is special in that the Durra compiler will provide a default value for it. The default value will be the full name of the process i.e., the concatenation of component names starting with the application task description name down through the chain of process names leading to the specific process.

References

[1] M.R. Barbacci and J.M. Wing. *Durra: A Task-Level Description Language.* Technical report, CMU/SEI-86-TR-3 (DTIC: ADA178975), Software Engineering Institute, Carnegie Mellon University, December, 1986.

[2] M.R. Barbacci, C.B. Weinstock, and J.M. Wing. "Programming at the Processor-Memory-Switch Level." *Proceedings of the 10th International Conference on Software Engineering.* Singapore, April, 1988.

[3] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock. *The Durra Runtime Environment.* Technical report, CMU/SEI-88-TR-18 (DTIC: ADA199480), Software Engineering Institute, Carnegie Mellon University, July, 1988.

[4] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, S.L. Baur, D.C. Bixler, M.T. Heins. *Command, Control, Communications, and Intelligence Node: A Durra Application Example.* Technical report, CMU/SEI-89-TR-9 (DTIC: ADA206575), Software Engineering Institute, Carnegie Mellon University, February, 1989.

[5] M.R. Barbacci and J.M. Wing. "A Language for Distributed Applications." *Proceedings of the International Conference on Computer Languages.* IEEE Computer Society, New Orleans, Louisiana, March, 1990.

[6] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock, "Application-Level Programming." *Proceedings of the 10th International Conference on Distributed Computing Systems*, May, 1990, Paris, France.

Index

- 10, 11

" 10, 11
* 11, 13
+ 10, 11
/ 11
/= 11
= 11
> 11
>= 11
{ 7
| 7
} 7
" 7

ActualParameterPart 17
application description 2, 5
ArithExpression 11
Attribute 23
AttributeDescPart 23
AttributeName 9
AttributeSelPart 23

BaselineComponents 29
BehaviorPart 21

channel description 4
channel implementation 4
ChannelDescription 14
ChannelName 9
ChannelSelection 17
cluster 2, 4
cluster manager 5, 6, 32
ClusterClause 27
ClusterComponentName 28
ClusterDeclaration 28
ClusterName 9
CompilationUnit 13
ComponentClause 27
ComponentConnection 30
ComponentDeclaration 27
ComponentName 9
CompoundTaskDescription 14

DeltaExcludeComponents 29
DeltaIncludeComponents 29
DestinationPortName 30

ENTER structure name 32
EXIT structure name 32
Expression 11
ExternalPortBindings 30

Factor 11
FormalParameter 14
FormalParameterPart 14
FromStructure 32
FunctionCall 10
FunctionName 35

GlobalAttributeName 9
GlobalComponentName 9
GlobalLinkName 9
GlobalPortName 9

identifier 9
identifier parameter 14, 15
integer 11
integer parameter 14, 15
IntegerExpression 11
IntegerLiteral 10
IntegerRange 11
IntegerValue 10
InterfacePart 19

link 4
LinkName 9

NestedStructureDeclaration 29
null port 31

Package_NameAttribute 38
ParameterName 9
ParameterType 14
port 4
PortBind 30
PortDeclaration 19
PortName 9
PREVIOUS structure name 32
Procedure_NameAtribute 38
process 4
Process_NameAttribute 39
ProcessName 9
ProcessorAttribute 37

real 11
real parameter 14, 15
RealLiteral 10
RealValue 10
Reconfiguration 32
ReconfigurationClause 27
Relation 11
RelOp 11

ScalarStructure 13
SimpleTaskDescription 14
SourcePortName 30
Specification 21
SpecificationName 9
SpecificationValue 21
string 11
string parameter 14, 15
StringLiteral 10
StringValue 10
StructureClause 27
StructureDeclaration 28
StructureName 9
StructurePart 27

task description 4
task implementation 4
TaskDescription 14
TaskName 9
TaskSelection 17
Term 11
ToStructure 32
TypeDeclaration 13
TypeName 9

UnionStructure 13

Value 10

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S CMU/SEI-91-TR-18	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-91-TR-18
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (City, State and ZIP Code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/AVS Hanscom Air Force Base, MA 01731
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESD/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (City, State and ZIP Code) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63756E; PROJECT NO.: N/A; TASK NO: N/A; WORK UNIT NO.: N/A

11. TITLE (Include Security Classification)
Durra: A Task-Level Description Language Reference Manual (Version 3)

12. PERSONAL AUTHOR(S)
Mario R. Barbacci

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM TO	December 1991	49

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Distributed programming, Distributed systems, task-descripton languages

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Durra is a language designed to support the development of distributed programming applications consisting of concurrent, large-grained processes devoted to specific pieces of the application. During execution time the application processes run on possibly separate processors, and communicate with each other by sending messages of different types across communication links. The application developer is responsible for prescribing a way to manage all of these resources, called a task-level application description. It describes the processes to be executed, the assignments of processes to processors, and the communication channels required to transmit messages data between processes. Durra is a task-level description language, a notation in which to write these application

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL John S. Herman, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7631 22c. OFFICE SYMBOL ESD/AVS (SEI)

descriptions.

This document is a revised version of the original reference manual[1]. It describes the syntax and semantics of the language and incorporates all the language changes introduced as a result of our experiences writing application descriptions in Durra.

A companion document, *Durra: A Task-Level Description Language User's Manual*, describes how to use the compiler and support tools.

1. M.R. Barbacci and J.M. Wing, *Durra: A Task-Level Description Language*. Technical reptort (CMU/SEI-86-TR-3, DTIC: ADA178975), Software Engineering Institute, Carnegie Mellon University, December, 1986.